Čas 1 - Uvod u C++. Strukture podataka u STL

- imenski prostori (namespaces) - služe
- ključna reč new kreira pokazivač na objekat (kao u Javi), s tim što moramo tada manuelno dealocirati objekte koje ovako napravimo
  - bez ključe reči new, iako pravimo objekat, to je više kao da se pravi promenljiva na steku, i ona će automatski biti dealocirana kada joj istekne doseg
  - u prvom slučaju je objekat null ako ne pozovemo konstruktor, a u drugom se default konstruktor automatski poziva
  - ono što znamo iz jave je ekvivalentno inicijalizacijom sa pokazivačem i ključnom reči new
  - kada se završi blok, za sve objekte koji treba da budu obrisani, poziva se **destruktor**
- metod - funkcija klase, funkcija koja se poziva nad objektom
- ključna reč **auto** se koristi da kompajler sam zaključi koji tip je potreban
- kada npr. ispisujemo clanove niza u petlji iteriranjem, svaki put ce se napraviti kopija tog clana. To nije dobro kada imamo objekte koji zauzimaju veliku memoriju, pa zato je dobro prenositi ih po referenci.
  - npr. void prikazi(student& st)
  - međutim, ovo ynaci i da mogu da ih modifikujem. Ako hocemo da zabranimo ovo, onda cemo reci `const int& number`

```
// Na kraju, najbolji format petlje izgleda ovako
for (const auto& number: numbers) {
    // ispisi broj
}
```

- Iteratori:
  - [c][r]begin()
  - [c][r]end()
  - `std::cbegin(vector)` - ovo je opštije, jer je obična funkcija, a ne metod/članica funkcija, pa je preporuka koristiti ovo umesto `.cbegin()`
  - r - reverse
  - c - constant
- accumulate iz biblioteke numeric <=> foldl, reduce
- std::cin je objekat klase std::istream. U toj klasi je definisano šta znači da li je neki objekat te klase true ili false, pa zato možemo raditi `while (std::cin >> number)`.
- Operator `>>` uzima argument `std::istream& cin` i number, izmeni ih, i onda vrati baš cin. Zato je objekat u while proveri i dalje `istream`, pa možemo proveravati da li je true ili false
- voditi računa kada se radi sa `while(std::cin)`, ne može se detektovati da je sledeći karakter EOF, već će se on prvo učitati, pa tek u sledećem prolazu će se petlja zaustaviti. To dovodi do

jedne više iteracije nego što smo planirali. Zato je bolje izvesti ovo na sledeći način:
`while(getline(std::cin, line))`

STL kolekcije

Primer generičke funkcije

```
template <typename T>
void show_range(T begin, T end) {
    for (auto iter = begin; iter != end; iter++) {
        std::cout << *iter << std::endl;
    }
}
```

Vektori

- postoji funkcija za proveru da li je container empty, i to je `vector.empty()` . Treba to koristiti, a ne `vector.size() == 0` , jer to nije ono sto semanticki zelimo
- `vector.insert(iterator/pokazivac na koji zelimo da ubacimo, sta ubacujemo)`
- `std::distance(std::cbegin(vector), std::cend(vector))` - distance vraca broj elemenata izmedju 2 pokazivaca
- `vector.pop_back()`
- `vector.erase(std::cbegin(xs) + 1)` - daj mi iterator na neki element i ja cu da ga izbacim
    - problem sa ovim je sto odmah pretpostavljamo da ovaj vektor ima bar dva elementa, pa je dobro to proveriti jednim if-om pre nego sto nesto ovako uradimo

Liste

- `std::list<int> xs {1,2,3,4};`
- dvostruko povezana lista, pa samim tim i odgovarajuce metode:
    - xs.push_back();
    - xs.push_front();
    - xs.pop_back();
    - xs.pop_front();
- xs.front() - returns a read/write reference to the data at the first element of the list (dakle ovo je obicna referenca, i mozemo je modifikovati)
- xs.reverse()

Skup

- poenta skupa je da nam eliminise duplikate

- stvari se cuvaju sortirano, u kolekciji koja je prirodno uredjena - stablo => ako hocemo da stavimo neku klasu u <> skupa, moramo imati implementirano poredjenje pre nego sto to uradimo
- `xs.find()`
- `xs.upper_bound(x)` - damo mu arg, i on pokusa da nam nadje prvi koji je strogo veci od njega
- `xs.lower_bound(x)` - prvi element veci ili jednak od x (ako taj element postoji, ofc ce vratiti bas taj element)
- `xs.lower_bound(x) - 1` - nalazenje elementa koji je strogo manji od x
  - s obzirom da ovo radi nad skupom, koji je uvek sortiran, ovo ce imati smisla

```
const auto x = 100;
auto finder_iter = xs.find(100);
if(finder_iter != std::cend(xs)) {...}
```

Mapa

- std::map<std::string, std::vector<std::string>>
- map<key, value>
- **mapa je zapravo niz uredjenih parova, gde je prvi clan para key, a drugi value**
  - std::pair<key, value>
  - bas zato se prvi element dobija sa first, a drugi sa second

```
std::map<int, std::string> studenti {
    {1, "Marija"},
    {2, "Veljko"},
    {5, "Tijana"},
    {10, "Marko"},  // delimiter je na kraju tokena, a ne izmedju tokena, sto je dobro
                    // ovde da bismo mogli da saltamo redosled elemenata (PPJ)
}
```

- kad iteriramo kroz mapu, tip iteratora je isti kao tip elementa mape, sto je par
- operator indeksiranja ovde, `studenti[3]`, pokusava trazenje tog studenta, a ako ga ne nadje, poziva se default konstruktor za vrednost (value u mapi), sto je u nasem slucaju std::string, pa ce rezultat poziva `studenti[3]` biti kreiranje novog elementa, sa kljucem 3 i praznom vrednoscu (to je default konstruktor za string)
- kada hocemo da dodamo element u mapu, moramo prvo konstruisati par koji cemo dodati. To mozemo raditi sa obicnim konstruktorom `std::pair<int, std::string> student {10, "Jelena"}`, a mozemo i funkcijom `make_pair`
- `studenti.insert(student)` - dodavanje u mapu
- medjutim, sta kada ubacujemo nesto sto vec postoji? Nista, samo nece biti ubacen. Medjutim, bilo bi korisno da to nekako programski znamo:

- `auto insertionStatus = studenti.insert(student)`
- find vraca `<iterator, bool>`
- ako je neuspesno ubacivanje u mapu, vraca se iterator na element koji je zabranio ubacivanje
- first je insertionPoint, second je insertionSuccessful
- `student.emplace(100, "Pera")` - konstruiše uređeni par sa ovim argumentima, i ubacuje ih u mapu studenti

Forward list

- za kad znamo da nam treba samo jednostrano povezana lista
- iterator `before_begin()`

Deque

Stvari iz novog standarda

- `const auto [insertionPoint, insertionSuccess] = studenti.insert(student)`
  - raspakivanje, kao kad u Pythonu uradimo `return [x,y]` , a onda to procitamo sa `x, y = ta_fja()`

Cas 2 - Dinamicka memorija

- podrazumevana vidljivost za strukturu je `public` , a za klasu `private` - to je u sustini jedina razlika izmedju klase i strukture u C++
- **lista inicijalizacije** - pri alokaciji memorije ce se odmah upisati odgovarajuce vrednosti, a nece morati ponovo da se pristupa memoriji, tako da je brze

```
point(double x, double y)
    : m_x(x), m_y(y) {}
```

- `std::to_string(nesto_sto_nije_string)`

```
// Odmah kreiraj i dodeli vrednosti
point p1(2,3);
// Napravi privremeni objekat, dodeli ga p2, pa ga obrisi. Optimizaatori ce vrv videti
point p2 = point(100,200);

// Objekat na hip memoriji
point* p3 = new point(2,3);

// ovo je OPERATOR, ne funkcija.
delete p3;
```

- postavljati default vrednosti za argumente da bismo imali default konstruktore. Ovo je dobro kada npr. pokusavamo da dinamicki alociramo niz objekata, a ti objekti se nikako ne mogu konstruisati bez argumenata.
  - `point* arr2 = new point[numOfPoints]`
- `delete[] arr2` je nacin da dealociramo ceo niz. Kada alociramo nesto sa `new .. []` moramo ga dealocirati sa `delete[]`
- **reference counting** - trivijalan algoritam automatske dealokacije objekata (kada je broj pokazivaca na memoriju 0, dealociraj je)
  - u Cpp-u imamo ovo, i ti pokazivaci koji ce biti brojani i sistem koji dealocira stvari se zove:
  - **`std::shared_ptr<T>`** - klasa omotac koja ce sve prosledjivati na svoj element koji je zapravo pokazivac
  - `std::shared_ptr<point> ptr(new point(2,3))`
  - za ovo nam treba zaglavlje **`#include<memory>`**
  - `std::shared_ptr<student> ptr = std::make_shared<student>(1,"Goku")`
  - **`auto ptr = std::make_shared<stundet>(1,"Goku")`** - ovo je preporuceni nacin za koriscenje ovoga
  - `ptr.use_count()` - staticka varijabla u klasi shared_ptr cuva broj referenci na ptr, a ovako joj mozemo pristupiti
  - voditi racuna o tome da ako saljemo ptr po vrednosti u funkciju, i tamo ispisujemo use_count, onda ce broj referenci bit iza 1 veci od ocekivanog
- **unique_ptr**
  - postoji tacno jedan pokazivac koji je odgovoran za vlasnistvo/brisanje/dealociranje objekta
  - postoji i **`make_unique`** - potpuno je ista prica
  - nullptr je konstanta koja govori da ovaj pokazivac trenutno ne postoji. Mnogo je bolje koristiti nullptr od NULL jer je NULL samo makro za 0.
  - unique_ptr nema konstruktor kopije (konstruktor dodele), pa ga zato izmedju ostalog ne mozemo prenositi po vrednosti kao argument funkcije (ali moze preko move semantike - vise o tome ispod)
  - kada funkcija dobije unique_ptr, ona je samo privremeno vlasnik tog pointera, ali je i dalje originalan pokazivac vlasnik tog pokazivaca.
  - `void f(std::unique_ptr<studnet>& ptr)` - gornji slucaj
  - ako hocemo da prenesemo vlasnistvo unique_ptr u funkciju kojoj ga saljemo, onda to mozemo uraditi koriscenjem **move** semantike
    - `show_student(std::move(ptr))` - tada ce se ovaj objekat dealocirati nakon sto se zavrsi funkcija show_student. Voditi racuna da onda moramo preneti unique_ptr po vrednosti u definiciji funkcije. U daljem kodu ce interni pokazivac unique_ptr biti nullptr.
- kada koristiti shared_ptr i unique_ptr?
  - unique_ptr treba koristiti samo za velike objekte i za one za koje nema smisla da mnogo objekata ima referencu na njih
- **streamovi**

    - `#include<sstream>`
    - `std::stringstream ss;`
    - koristi se sa `<<` , alternativa da sami konvertujemo brojeve svaki put sa `std::to_string`

RAII

- resource acquisition is initialization -
- `#include<fstream>`
    - `std::ifstream m_fajl = std::ifstream("ime_fajla.txt")`
    - `m_fajl.is_open()`
- `std::endl` - prazni buffer, a `\n` samo ispisuje novi red

Cas 3 - Objektno-orjentisano programiranje, UML dijagrami (deo 1)

- podrazumevane vrednosti za konstruktore cemo drzati samo u .hpp fajlu
- svi geteri treba da budu `const` , jer ne menjaju unutrasnje stanje klase

```
// Ovim sprecavamo cirkularno ukljucivanje incudova
#ifndef FRACTION_HPP
#define FRACTION_HPP

#endif



Fraction operator+(const Fraction& f) const {
    return implementirano_sabiranje_razlomaka;
}

// binarni minus
Fraction operator-(const Fraction& f) const {}

// unarni minus
Fraction operator-() {}

bool operator==(const Fraction& other) const {
    // treba proveriti da li su skraceni, ali podrazumevacemo da jesu
    return m_num == other.m_num && m_den == other.m_den;
}

bool operator!=(const Fraction& other) const {
    return !(*this == other);
}
```

```
// ovo se pise u okviru klase, iako je ovo obicna funkcija, pa zato ima friend
// friend nam omogucava da imamo pristup privatnim poljima klase
friend std::ostream& operator<<(...);
friend std::istream& operator>>(std::istream& in, Fraction& value);

// cpp fajl
std::ostream& operator<< (std::ostream& out, const Fraction& value) {
    // ovim narusavamo enkapsulaciju (friend svesno to radi), da bismo
    // sacuvali na performansama
    return out << value.m_num << "/" << value.m_den << std::endl;
}

std::istream& operator>>(...) {
    char delimiter;
    return (in >> value.m_num >> delimiter >> value.m_den);
}
```

- ekspilictna i implicitna konverzija:

```
// operator za kastovanje. dovoljno je odluciti se za jedan double i ideja
// je da to bude drugi
// double operator double() const {}

operator double() const {
    return 1.0*m_num/m_den;
}

operator bool() const {
    return m_num != 0;
}
```

- inrementiranje (postfiksna i prefisknaa)

```
// Prefiksna inkrementacija
Fraction& operator++() {
    const auto tmp = Fraction(1,1) + *this;
    *this = tmp;
    // moglo je i samo m_num+=m_den
    return *this;
}

// Postfiksna inkrementacija
// Dodavanjem int u parametre cemo omoguciti da kompajler razlikuje ovo od prefiksnog.
Fraction operator++(int) {
    Fraction tmp(m_num, m_den);
    ++(*this);
    return tmp;
}
```